CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

Subject: **Final Report - Software Reuse in Real-Time Environments**

CIN: **C04 092LK 0001 00**

8 DECEMBER 1989

Software Reuse in Real-Time Environments

by

T. P. Baker
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019

for

U.S. Army HQ CECOM, Center for Software Engineering

October 9, 1989

Contract No. DAAL03-86-D-0001
Delivery Order 1135
Scientific Services Program

Contents

Summary

This report addresses two directions in which reuse of Ada software might be encouraged for realtime applications. These two directions are:
(1) improvements to the Ada language through the Ada 9X revision process;
(2) development of more common (i.e. standard) package interfaces.

Given that the design of the Ada programming language predates the slogan "software reuse", Ada supports software reuse rather well. However, there remain some ways in which Ada could be improved. Most serious of these is the lack of uniformity across Ada implementations of certain features important to realtime applications, like storage dynamic management and task scheduling. Also important are a few gaps in Ada's representation-specification mechanism.

In the area of common interfaces, runtime environment (RTE) interfaces are especially important to realtime applications. Three kinds of RTE interfaces may be distinguished: application-RTE; compiler-RTE; RTE-RTE. Agreement on each of these can further software reuse.

The Model Runtime System Interface (MRTSI) [6] is a proposed model for a standard compiler-RTE interface, developed by the ARTEWG. If successfully adopted, such a standard could foster reuse of Ada runtime system implementations across compilers, and reuse of compilers across runtime system versions.

The IEEE Portable Operating System Interface for Computer Applications (POSIX) Ada binding (1003.5) and the POSIX realtime extension (1003.4) are examples of proposed standard application-RTE interfaces. If successfully completed, these interfaces could lead to more reuse of applications across computer systems, and promote reuse of components across applications. Both of these interfaces could foster reuse in a wide class of Ada applications, including compilers and software tools.

The POSIX realtime extension could especially benefit realtime applications, by providing features missing from Ada. While supporting realtime applications within a general purpose operating system like POSIX is bound to exact a significant compromise in performance, there are benefits that may sometimes outweigh the performance loss. Potential benefits include being able to combine cooperating realtime and nonrealtime applications in a single system, and being able to prototype and debug realtime applications in a development environment.

Both the MRTSI and POSIX 1003.4 and 1003.5 activities have not yet reached implementation, but promise significant rewards. More effort is needed to see these standards through to completion, and to insure that they provide the features needed by realtime applications.

The area of RTE-RTE interfaces looks equally promising. Standard interfaces for RTE components that typically need modification to fit particular realtime applications, such as I/O and clock drivers, would assist in the reuse of realtime application code across Ada implementations, and the reuse of Ada implementations across different application hardware configurations. So far, there does not yet appear to be any organized activity focussed on this area, but it deserves attention.

Several issues raised by this study appear to deserve further attention. These are detailed in the conclusion to this report.

Acknowledgements

The preparation of this report was funded by the U.S. Army Communications Command (CECOM), through the U.S. Army Research Office and Battelle Laboratories. It also draws on background work supported by the Florida State University and the Office of Naval Research, in the area of high-level language support for realtime software systems.

Ideas expressed in this report owe much to discussions with the members of IEEE POSIX Ada binding working group (1003.5), the Ada Runtime Environment Working Group (ARTEWG) of the Special Interest Group on Ada (SIGAda) of the Association for Computing Machinery (ACM), and participants at several workshops on realtime Ada issues attended by the author.

1 Introduction

1.1 Scope.

This report examines the roles of several factors affecting the potential reuse of Ada software for realtime applications. More specifically, it examines the roles of the Ada language and common package interfaces. The Ada language is evaluated with regard to how well it meets several requirements for supporting reuse, and how it could be revised to support reuse better. The concept of common "bridge" package interfaces is introduced as a technique for enabling more software to be reused. Three specific common package interfaces are examined in detail. These are: the MRTSI; the POSIX Ada binding (1003.5); the POSIX realtime extension (1003.4). These interfaces are considered with respect to their individual potential for promoting software reuse, their interactions, and their potential for supporting and complementing Ada for realtime applications.

1.2 Approach.

Information in this report has been gathered from several sources, including: the experience of the principal investigator as a developer of an Ada compiler and two Ada runtime systems; participation in the ARTEWG and the POSIX (1003.4 and 1003.5) standards groups; the International Workshops on Real-Time Ada Issues (1987, 1988, 1989), the Ada 9X Requirements Workshop (1989); experience with a partial prototype implementation of the POSIX realtime extension.

Of the three interfaces examined in detail, the least is known about the POSIX realtime extension. The C-language binding of the POSIX realtime extension is still in draft, no work has yet been started on an Ada binding, and there are no complete prototype implementations. In contrast, the MRTSI exists as a complete document, including Ada package specifications, and has been prototyped. The draft POSIX Ada binding is also fairly complete, and covers functionality for which C-language implementations have existed for many years.

In order to gain a better understanding of the POSIX realtime extension, as part of this project, work was started on a prototype implementation. This work involved modification of the Sun Microsystems UNIX[1] operating system (written in the C-language) to support certain features of the draft POSIX realtime extension, development of a very crude Ada binding for these features, and experimentation with them.

1.3 Background

What is software reuse? "Software reuse" is currently a popular slogan in the Ada and software engineering community. It denotes a tactic for software development: to look carefully for ways to use existing software before dashing off to write more.

[1] UNIX is a registered trademark of AT&T.

Kinds of reuse. The tactic of software reuse can be applied at any scale and any level of abstraction. The scale of reuse can range through: (1) using all of an existing piece of software, instead of developing a new one; (2) modifying or upgrading an existing piece of software, retaining most of the existing design and code; (3) using fragments of existing software in a new design. If only fragments are reused, the fragments may range from concrete code, such as subprograms and data types, to abstractions such as scheduling policies and mathematical models.

Benefits of reuse. Software reuse can pay off in several ways, including increased reliability, reduced development effort, more predictable size and performance, reduced code size, and improved readability. Reliability is increased because reused software has been previously tested; therefore it should be more reliable than new software. Flaws are likely to have been discovered, and corrected. Development effort is reduced when reusing existing software by skipping the steps of detailed component design, coding, and debugging. Size and performance requirements (processor time and memory) of reused components are known; thus, system-level size and performance problems can be detected earlier than with new software. Total code size of a large system may be reduced by sharing software components across subsystems. Readability is improved if shared components are well designed, since there are fewer total components.

Limitations to reuse. The degree to which software reuse can be effective is limited by several factors. Differences in real requirements which may prevent code developed for one project from being reused on another. There may also be confusion between arbitrary design decisions and real requirements, so that new software appears to be required, even though existing code could be used. Cultural differences between programmers may make it hard for a programmer to understand code or documentation that another has written, or to see how it can be used in a new program. Even where programmers are trained in the same way of thinking and writing, it may take more time to understand existing software well enough to decide whether and how to reuse it than to write new software. Programmers also tend to have a psychological drive to write new software. Administrative policies can seriously discourage reuse; for example, by measuring productivity in lines of code or not allowing charges for time spent reading existing code. Given large volumes of potentially reusable code, information retrieval becomes a problem. Most of these limiting factors are hard to address technically, since they involve primarily political, social, and psychological issues.

This paper addresses two factors which appear more amenable to technical solutions. These are (1) the limitations imposed by programming languages and (2) the lack of standard interfaces. It seems to be a good time to think about these two subjects, since the Ada 9X language-standard revision project offers a unique opportunity to improve the Ada programming language's support for software reuse, and several secondary standards activities that are currently under way offer opportunities to develop other needed common interfaces.

1.4 Organization.

Section 2 examines the the Ada programming language with respect to support for software reuse. Section 3 discusses the importance of common Ada interfaces for software reuse. The

three sections after that examine specific interfaces with potential impact on reusable realtime applications: Section 4 deals with the MRTSI interface, Section 5 deals with the POSIX Ada binding, and Section 6 deals with the POSIX realtime extension. Section 7 is the conclusion.

A more detailed report of the work on prototyping portions of the POSIX realtime extension is provided as Appendix A and a realtime demonstration program constructed for the realtime extensions is described in Appendix B.

2 Ada

In a primitive sense, every programming language may be viewed as a system of reusable software components. The language defines a set of primitive components and a notation for combining these components. The programmer uses this notation to specify how the language-defined components are combined in each particular program. The compiler puts together the program, following this specification. It translates each instance of a language feature into machine instructions, according to a (reusable) translation template, which is designed by the implementor of the language feature. The translation template for some features may be very concrete (e.g. for literals and arithmetic operators) and for others (e.g. data types, subprograms, and tasks) it may be rather abstract.

The more interesting kind of software reuse involves larger, programmer-defined components. For a language to support this kind of reuse it must be *extensible*. That is, programmers must be able to define their own coding templates, and the notation by which they can combine instances of these templates to construct new programs. Languages vary quite widely in the degree to which they support this level of reuse.

Though supporting software reuse was not one of the explicitly stated design goals of the Ada programming language [3,4], Ada does a very good job of it. This is not surprising, since the designers did consciously attempt to support application portability, and programming techniques that are closely related to software reuse, including abstraction and bottom-up development. Some specific Ada features that support software reuse include packages, strong typing, and generics.

To see more precisely how Ada supports reuse, and how it might be revised do better, six key issues will be examined. These are: (1) partitioning; (2) importation; (3) parameterization; (4) information hiding; (5) component extensibility; (6) portability (i.e. independence of software from the target machine architecture and the language implementation).

2.1 Partitioning

A programming language should provide some way of partitioning programs into potentially reusable modules. Ada's separately compilable units, in particular packages, provide this capability.

2.2 Importation

A language must provide an importation mechanism; i.e., a way of incorporating existing software components into a program by reference, without introducing name conflicts. Ada's library units and context clauses serve this function very well. Name conflicts may be resolved via operator overloading, qualified names, and renaming declarations. This set of importation and renaming mechanisms appears to be functionally complete.

Ada's renaming mechanism does have some ugly aspects. For example, it is annoying that

renaming declarations for constants and types must masquerade as variable and subtype declarations. It can also be annoying that renamings are considered distinct from the original declarations for the purpose of checking conformance of subprogram specifications and type discriminant-parts. Fortunately, one can work around these inconveniences.

2.3 Parameterization

To support software reuse a language should provide a parameterization mechanism. Parameterization allows a component to be reused in a variety of contexts, by allowing the specification of certain details to be deferred to each context where the component is used. Ada generic units, which were primarily conceived as a mechanism for reducing code size ([4] p. 235), also permit compile-time parameterization. This mechanism goes a long way toward supporting software reuse. Still, there are several respects in which it is not complete.

Gaps in generic parameters. It is instructive to compare the language rules for generic parameters with those for renamings, since the parameters of a generic instantiation serve an analogous function to the imported declarations of a conventional program unit. For full programming generality, one would hope that anything that could be named could also be specified by a generic parameter, just as anything that can be named can be imported and renamed. Consider the following table:

Entity	Renaming	Generic Formal
discrete type	ok	ok
integer type	ok	ok
fixed pt. type	ok	ok
flt. pt. type	ok	ok
access type	ok	ok
record type	ok	→ private, limited private
task type	ok	→ limited private
private type	ok	ok
ltd. priv. type	ok	ok
variable	ok	ok
constant	ok	ok
univ. constant	ok	not supported
exception	ok	not supported
package	ok	not supported
subprogram	ok	ok
entry	→ procedure	→ procedure
task	ok	ok

As can be seen from the table, there are gaps in the generic parameterization mechanism for record types, task types, universal constants, exceptions, and packages. While there are understandable reasons for these gaps, they are not completely necessary and they do limit software reuse. Let us consider how these gaps might be closed through revisions to the Ada language.

Universal constants. The restriction on universal constants means that the ranges, precisions,

and representations of numeric types declared within a generic software component are fixed at the time it is written. This restriction is probably a consequence of the intent that generic bodies and instantiations can be checked for correctness independently ([4] p. 266). In particular, this restriction helps to insure that values of static expressions within the generic body do not depend on the values of generic parameters. It is certainly a serious limitation on software reuse, but the cost of relaxing it is also high.

Exceptions. The restriction on exceptions is probably also a consequence of the principle of independence of bodies and instantiations (cited above). Without this restriction, generic formal exceptions could provide aliases for one another and for directly named exceptions in handlers, making it impossible to check for overlapping cases until instantiation. This could be solved by allowing generic formal exceptions only in **raise** statements, or by checking for aliasing in a fashion similar to the way checking is done for similar requirements on generic formal types (see next paragraph).

A precedent. The case for allowing generic universal constants and exceptions is not completely hopeless. There is at least one precedent for violating the principle of idependence of generic bodies and instantiations. Notably, certain usages of a generic formal type within the generic body require that the corresponding actual type in each instantiation must not be an unconstrained array type or a discriminated type without default values for the discriminants. (This is very hard to check, especially if the use is in a separately compiled subunit that is added after the generic body and instantiations have been compiled.) Unless this rule is changed, there is a precedent for adding other such cross-dependences.

Types with components. Record types, task types, and packages have one common attribute: they include exported components, which must be taken into account in matching formal with actual generic parameters. This is not an insurmountable problem. A consistent notion of matching can be defined recursively. Suppose a generic formal record type declaration is allowed, with components following the syntax of formal object declarations. A generic actual parameter type can be said to match this formal type if each component of the formal type has a specified corresponding component with a matching type. This is illustrated in the example below. (Notice that this example makes another extension, to allow incomplete type declarations in the generic formal part.)

```
generic
  type REC;
  type POINTER is access REC;
  type REC is
    record
      NEXT: POINTER;
    end record;
package LISTS is ...
end LISTS;
```

```
type NODE;
type LINK is access NODE;
type NODE is
  record
    SIBLING, PARENT: LINK;
    I:   INTEGER;
  end record;
package SIBLING_LISTS is
  new LISTS(POINTER=> LINK,
            REC=> NODE (NEXT=> SIBLING));
```

Unlike the previous two cases, this extension does not appear to conflict with the goal of checking the correctness of generic bodies independently of the instantiations. The chief drawback to such an extension seems to be that a compiler could not share object code between different instantiations. The matching rule for formal private types with discriminants ([1] 12.3.2 (3)) offers a precedent for this extension and suggests a simpler and more restrictive matching rule, based on positional correspondence of component declarations, which might also permit more

code sharing.

An analogous recursive matching rule can be defined for components of packages and entries of task types.

Subprogram parameters. One of the few ways Ada takes a step backward from other languages in its support for software reuse is by not having subprogram-types — i.e., not allowing variables and subprogram parameters which take subprograms as values. This rules out "call-back" interfaces, which have been used by reusable communication protocols and windowing toolkits in other languages. There is no good reason not to support procedure types in Ada, since compile-time type checking is possible. Of course, the values of variables and out or in out parameters of procedure types should probably be limited to subprograms declared as library units or immediately within library packages, in order to prevent calling a subprogram from outside its scope. There would be no significant implementation problem, since code-sharing implementations of generics already need a similar mechanism to handle generic formal subprograms.

2.4 Information hiding

A programming language should enforce information hiding. Simple, precisely defined interfaces make components and their interactions easier to understand, and therefore easier to reuse.

Ada takes a big step in this direction, by separating program units into specifications and bodies, and providing private and limited private types. Unfortunately, the separation of interface specification from implementation is incomplete. Package specifications provide both the information needed by users and information the language designers felt a compiler would need to perform efficient separate compilation. This is bad from the point of software reuse, since it means package specifications may need to be modified to support changes in the corresponding bodies.

It is ironic that the reason for requiring implementation information in package specifications is largely obviated by other language complexities. Most of the semantic problems associated with deferring full declarations of private types and constants to the package body are already encountered with incomplete type declarations. Moreover, Ada is sufficiently complicated that some compilers already defer final code generation until package bodies are seen, in order to produce more efficient code. It therefore appears the information provided in the package private part is not essential.

2.5 Component extensibility

To support software reuse a language should provide a mechanism for extending the functionality of existing components. Ada does not do this very well.

For example, consider the POSIX Ada binding. This is a standard to which a series of extensions will be added as time goes on. There are some logically private types for which operations are defined in several basic POSIX packages. More operations on these types will be defined

in packages that will be added to the standard in the future. Implementing some of these operations may require access to the full type representation. It is essential that such extensions not require modification to the basic POSIX package interfaces.

Some other languages (e.g. Smalltalk) provide an "inheritance" mechanism to allow this sort of reuse. With inheritance, an extension package can inherit the types and operations of the package it extends, so that only the new operations require new code. Such languages mostly do not enforce information hiding.

If information hiding is not important it is easy to write inheritable packages in Ada. No private types are used; the full declarations of all types are included in the package's visible part, so they can be visible in the bodies of packages that extend it.

If information hiding is important, it appears the only way to solve this problem in Ada is to use unchecked conversion within the extension package body. (Such a solution is sketched below.) Whether this solution is acceptable appears to be a matter of taste. Information hiding is still preserved for the users of the packages, and "`with UNCHECKED_CONVERSION;`" in the context makes it clear that the extension package body is doing something that violates information hiding.

```
package CORE is
  type T is private;
  ... -- some operations on T
private
  type T is ... ; -- some type definition
end CORE;
```

```
with CORE;
package EXTENSION is
  function F(X: CORE.T) return CORE.T;
  ... -- more operations on T
end EXTENSION;
```

```
with UNCHECKED_CONVERSION;
package body EXTENSION is
  type XT is ... ; -- duplicate of CORE.T's full definition
  function FORCE is new UNCHECKED_CONVERSION(CORE.T,XT);
  function FORCE is new UNCHECKED_CONVERSION(XT,CORE.T);
  function SOME_OPERATION(X: XT) return XT is
  begin ... -- something that requires visibility of full type XT.
  end SOME_OPERATION;
  function F(X: CORE.T) return CORE.T is
  begin return FORCE(SOME_OPERATION(FORCE(X)));
  end P;
end EXTENSION;
```

It is not clear that this problem could be solved much better, even if one were willing to change Ada radically. In fact, there seems to be an inherent conflict between information hiding and extensibility. One compromise, adopted by C++, is to distinguish two classes of importation: "hostile" users are only allowed to import declarations from a package's visible part, while "friendly" packages are allowed to import declarations from its private part and body.

This hostile/friendly distinction might be added to Ada, perhaps by adding a new attribute and extending the context clause to semantics to allow a package body to be imported by friends (`with` *package_name*`'body;`), but it is not clear whether this is much better than `with UNCHECKED_CONVERSION;`. More protection could be added by requiring the core package specification to explicitly name all the friendly packages, but then this would defeat our goal of being able to reuse and extend the package without modifying its specification.

If a package is generic, extending it is still more difficult, since each instantiation of the extension must match up with a compatible instantiation of the core package. While the unchecked conversion technique may work between instantiations with identical formal parameters, it is risky in general, due to the complex ways compilers sometimes translate generic instantiations.

2.6 Portability

Portability is an important aspect of software reusability. To be portable, application code must be both independent of the target machine architecture and the programming language implementation. The ideal of portability is to be able to take a software component compiled with one compiler and tested on one machine, compile it with a different compiler, run it on a different machine, and obtain equivalent results.

Writing portable code requires two things of the programming language: (1) it should permit necessary hardware dependencies to be isolatable; (2) other software components, which do not interface with hardware devices, should be perfectly portable. Ada partially meets both of these requirements, though it falls short of the ideal.

Machine independence. For realtime systems there is certainly some limit to machine-independence. Software that must interface directly to specific application hardware devices, as is typical in hard realtime embedded computer applications, will need to be modified when reused in a different hardware environment. However, if such hardware dependencies are encapsulated within a few small components, the rest of an application may be machine-independent.

Ada features such as representation clauses and machine-code inserts permit some hardware dependencies to be specified. Careful use of package interfaces can isolate such machine dependencies from the rest of the program. However, there are some gaps in the coverage of Ada's machine-specific programming mechanisms, and Ada implementations do not support these features uniformly. Working around a compiler restriction, such as not allowing hardware interrupts to be connected to task entries, is likely to force software design changes that cannot be kept local. Local machine-dependencies thus tend to become global compiler-dependencies.

Implementation independence. The precision of the Ada definition and the rigor of validation testing make it easier to write implementation-independent components in Ada than in other languages. However, there remain quite a few implementation-dependencies.

CECOM has produced a new version of the ARTEWG Catalog of Ada Runtime Implementation Dependencies [8], which contains 155 pages. (This subject is also addressed in [9].) Most of these dependencies are not likely to seriously limit portability, since one can program to avoid them. However, some of them cannot be avoided. Let us look at two examples of such critical implementation-dependencies.

One example of an Ada implementation-dependency is the predefined numeric types, like INTEGER and FLOAT. Since Ada does not specify the range or precisions of these types, code that uses them is typically not portable. A careful programmer can mostly work around this

problem by using only numeric types with explicitly defined attributes. One awkward detail is that there are a few contexts where the standard type INTEGER is required, such as the operator "**", and fixed-point "*" and "/". Another weakness of this workaround is that the resulting proliferation of numeric types makes reuse of components more complicated, due to extra type conversions. Since agreement on standard primitive types is one of the foundations of reusable software, Ada should be revised to generalize the operators that are presently only defined on standard INTEGER and to provide some standard names for numeric types of known range and precision.

Task scheduling is another example of a serious implementation dependency. One of the original motivations for including concurrent programming in the Ada language was to eliminate dependence on an application-specific executive. Such dependence on features of a particular executive has been cited as a major obstacle to software component reuse in realtime systems. Ada failed in this respect, since it has no precisely defined task scheduling policy. Realtime applications with hard scheduling requirements currently must rely on features of a specific vendor's Ada compiler or runtime environment.

This lack of standardization poses a portability problem not only for entire applications, but also for smaller components which may be intended for reuse. For example, consider a package which includes operations that read and update data stored within the package body. If there is a possibility that more than one task may use this package, the read and update operations must be treated as critical sections. The only way to do this in Ada is by rendezvous with a "server" task. Whether the server task solution is efficient enough to be usable for hard realtime applications is likely to depend on whether the Ada implementation can recognize such a task and replace the rendezvous by simple semaphore operations. Also, whether or not the efficiency of rendezvous is important, differences in the implementation of task priority are likely to cause portability problems for such a server task.

Ada does not require implementations to support any specific range of task priorities. At the extreme, this range may be null. In this case, or if the programmer does not specify a priority for a task, the implementation may use any scheduling policy it chooses. Thus, tasks with no defined priority may be treated as if they have maximum priority, minimum priority, or some dynamically adjusted priority. Moreover, there is no defined policy for sharing a processor between tasks of equal priority. Thus, if a task has no defined priority or has the same priority as some other independent task there is no guarantee that the task will ever execute. This seems to mean that portable code must assign a distinct priority to each task; but then this leads to new problems. The language implementation may not support enough levels of priority. Even if there are enough priorities, the code of reused components with internal tasks must be modified for each application, to adjust the priorities of the internal tasks to fit within the framework of the other tasks in the application.

One ostensible reason for Ada not defining a rigid task scheduling policy is that different applications may require different policies. If so, a standard way could be provided for an application to specify its task scheduling requirements, similar to the ways Ada currently provides for specifying other implementation details like the addresses and sizes of variables. This would reduce implementation dependence. Moreover, given availability of such user-controllable options, a default scheduling policy could be defined. Those applications for which the default policy is adequate would then be guaranteed portable.

There are quite a few other serious Ada implementation dependencies. Several of them are explored in [15]. Among these, some of the most critical for realtime applications are in the areas of dynamic storage allocation and reclamation, and of support for using shared memory to communicate with hardware, other programs, or an operating system (especially the treatment of **in out** parameters, and the pragma SHARED).

In general, Ada would be better for writing portable code if each of the semantic details that are now implementation-dependent were either specified precisely by the language standard or put under direct programmer control.

2.7 Prospects for software reuse with Ada

It is certainly practical to reuse software components written in Ada. Among standard high-level languages, Ada probably supports reuse better than any other. At the same time, there remain several ways Ada could be improved in this regard. Some such improvements may be expected as part of the Ada 9X language revision process; but for now, programmers designing for software reuse need to be very conscious of Ada's limitations.

For realtime applications, Ada's most serious limitations are its implementation dependencies (including some common implementation deficiencies). Implementation dependencies that do not affect reusability for general purpose applications can be critical for realtime applications. Reducing these implementation dependencies, and replacing them by programmer-selectable options where necessary, is very important.

For now, it makes sense to use Ada's strengths to compensate for its weaknesses. Ada's greatest strength for software reuse, whether for realtime or other applications, is its support for abstract interfaces – specifically packages. Package interfaces can separate non-reusable code from reusable code, by isolating the non-reusable code within the bodies of a few packages. This enables the reuse of other components.

3 Common interfaces

Importance of agreement. Common interfaces are an essential part of software reuse. Every reusable component has an interface – a set of conventions to which users of that component must adhere. Agreement on these interfaces is critical to reuse of components.

The Ada language standard defines only a few component interfaces. Reusable software must depend on other interfaces. Ada defines the bottom level of interfaces, between syntactic components (e.g. expressions, statements) of a program. It also defines a few higher-level interfaces, like the packages `CALENDAR` and `TEXT_IO`. Any others must be defined outside of the Ada language definition. These may be defined *ad hoc* or by standards.

Any group of users can agree on standard interfaces, and thereby promote reuse within that group. The more widely the interface is agreed upon, and more precisely it is documented, the greater the impact on reuse. The development of standard Ada interface specifications for frequently-needed components is therefore important to promoting software reuse.

Component dependence, and bridging components. Reuse of software components requires recognition of dependence between components. Let us say a component, A, *depends* on another component, B, if A cannot be used without B, and changes to B may require changes to A.

Reuse requires adherence to common interfaces on both sides of the reused component. On the one side, an application that depends on a reused component must meet the interface requirements of that component. For example, if the component is an Ada package and the application calls a procedure in that package, it must adhere to the parameter type requirements of the procedure (as well as any other constraints, functional or semantic, imposed by the designer of the package). On the other side, if the reused component depends on other components it must adhere to the interface requirements of those components. For example, if the body of a reused package calls a subprogram of another package, it must adhere to the interface requirements of that other package.

Based on the concept of dependency, Ada software components may be classified as *non-reusable*, *reusable*, or *bridging*, as follows:

1. **Non-reusable component.** Neither the specification nor the body of such a component is expected to be reused, perhaps because it depends on peculiar features of the language implementation, the application, or the application's hardware environment.

2. **Completely reusable component.** Both the specification and body of such a component is expected to be reusable. Neither the body nor specification of such a component may depend on a non-reusable component.

3. **Bridging component.** The specification of such a component is expected to be reused, but alternate bodies may be required to fit different language implementations or hardware environments. The body may depend on a non-reusable component, but the specification may not. Bridging components enable other software components to be reused,

by separating them from dependence on nonreusable components. The packages of the POSIX operating system interface and the ARTEWG CIFO[7] are examples of bridge components.

Standard package interfaces can support software reuse in two ways:

1. Directly. A standard package may be a completely reusable component, in which case it may be incorporated directly into a new program (specification and body).
2. Indirectly. The standard package may be a bridging component, in which case it may enable other components (that depend on it) to be reused. The body of such a standard package may need to be rewritten for different environments, but the applications of the package will not need to be changed.

3.1 The need for secondary standards

The need for standards for common Ada package interfaces outside of those required by the Ada language definition, has been raised repeatedly. Such standards are typically called *secondary standards* because they are based on the Ada language standard, but are not part of it. A virtue of a secondary standard is that since it is not be part of the Ada language it does not impose a burden on every compiler vendor in the same way that a new language-defined standard package would. Developing and maintaining an implementation of a secondary standard is a job for an application programmer or a third party. The cost need only be borne by applications where the package is used. Another virtue of secondary standards is that they may be developed and revised in a more timely fashion than the Ada language.

Several candidates for secondary standards have been brought up publicly. Two examples are the ARTEWG CIFO and MRTSI. The International Workshop on Real-Time Ada issues proposed several language extensions that might be suitable as secondary standards, including a package of low-level tasking operations[14]. The Ada 9X Requirements Workshop found a need for secondary standards, especially in the area of input and output, citing several examples of existing standards for which standard Ada interfaces are required, including SQL, LU6.2, GKS, and PHIGS[15]. Work is under way on secondary standards for Ada in several other areas, including numerical functions and operating system services. Work also seems to be needed in the area of data communications and networking.

Realtime applications especially need secondary standards to provide features that are missing from the standard Ada language, such as control over task scheduling. These features must be provided by an underlying runtime environment, which may involve one or more of the following: an operating system; the Ada language implementation's runtime system; application-specific runtime support code.

3.2 Ada runtime environment interfaces

Three kinds of Ada RTE interfaces may be distinguished. These are: (1) application-RTE interfaces; (2) compiler-RTE interfaces; (3) RTE-internal interfaces. These are all examples of

bridging component interfaces.

An application-RTE interface enables an Ada application program to obtain services from the underlying RTE. If there is more than one layer of RTE, as when an Ada language implementation is built on top of a conventional operating system, such an interface may provide visibility of more than one layer.

Common application-RTE interfaces are important for promoting portability and reuse of code. Unless such interfaces are standard, portions of application code that invoke RTE services will need to be rewritten when moved between RTE implementations.

The ARTEWG CIFO packages are an example of application-RTE interfaces which gives access to the Ada RTE. The purpose of these interface packages is to provide functionality which is needed by realtime programs but is not provided in a standard form by the Ada language. For example, one package provides programmer-controlled periodic and event-driven task scheduling.

POSIX is an example of a standard application-RTE interface, to an operating system layer below the Ada RTE. Such a standard operating system interface can increase the reusability of application programs which need to make use of operating system services, including Ada compilers and software development tools. The benefits of such an interface may also extend to realtime applications.

A compiler-RTE interface enables code generated by an Ada compiler to obtain services from the underlying RTE. These services are invoked implicitly by compiler-generated code for the implementation of language constructs that require runtime support, such as dynamic storage allocation, task creation, and rendezvous. An example of such an interace is the ARTEWG MRTSI.

Standard compiler-RTE interfaces are most important to compiler vendors who must maintain compiler and RTE versions for many different hardware environments. Without heavy reuse of components, this would not be practical. The principle of common compiler-RTE interfaces is therefore well established within organizations developing Ada compilers.

Common compiler-RTE interfaces are also important to application developers in the realtime embedded domain, chiefly because such application programmers need to tailor and maintain project-specific versions of an Ada runtime environment. Embedded realtime applications typically need to be directly involved in some hardware-dependent operations; for example, timekeeping and interaction with I/O devices such as sensors and actuators. Worse, the hardware environment tends to be different for each application, and to evolve during the application's lifetime. It is therefore necessary for the Ada RTE to be tailored to the application. Tailoring of the Ada RTE may also be necessary to provide missing features, or to improve performance.

It is often impractical to contract with the compiler vendor to perform such RTE work. Commonly cited reasons include time delays due to contractual red tape and vendor workload, and cost. If an application builder must take on the burden of maintaining a special Ada RTE, a stable well-documented RTE interface is needed. Moreover, if the reason for this special RTE is to provide extensions missing from standard Ada, portability of application components may require supporting versions of this RTE on several machine architectures. Commonality in RTE

interfaces between compilers can make this much more practical.

An RTE-internal interface is the kind of interface by which one RTE component obtains the services of another. Such interfaces have not yet been well publicized, but examples can be found in the proprietary literature of compiler vendors, and in [10,11,12]. If a C-language binding of the POSIX realtime extension is used to implement Ada tasking, this would also be an RTE-internal interface. (Portions might even end up as part of the compiler-RTE interface.)

Common RTE-internal interfaces may prove more effective than common compiler-RTE interfaces in solving most tailoring problems. It appears that some of the Ada RTE components that most often need tailoring may be isolated from the rest of the RTE, so that the rest of the RTE may be reused without change. These components include the interface to the hardware time-keeping function, dynamic memory allocation, resource management, and task scheduling. Again, adherence to common interfaces for such components is well established within individual organizations that develop compilers, out of economic necessity, but it has not received much public exposure.

4 MRTSI

4.1 Background

The Model Runtime System Interface (MRTSI) is a document describing a model interface between the code generated by an Ada compiler and the Ada RTE. This interace isolates from the compiler that portion of the Ada RTE which is responsible for supporting concurrent execution of tasks, and isolates this portion of the RTE from the compiler-dependent portions of the RTE.

The MRTSI interface was published in the ACM SIGAda newsletter *Ada Letters*, in January 1989 [6], after five major revisions, starting from an original draft produced in the summer of 1987. Input was provided by twenty-five individuals with experience writing Ada compilers, Ada runtime systems, operating systems, and realtime application programs. Some of this input was provided directly, at meetings, and some of it was provided in the form of written reviews received in response to a mailing of Version 1.4 to known Ada compiler developers. Versions 1.3 and 1.5.1 were prototyped and tested in simulation, but no commercial Ada compiler adheres to this interface. Projects for larger scale implementation and testing of the MRTSI or a derived interface are under discussion. In particular, the U.S. Air Force has issued an announcement requesting proposals for a common Ada runtime system demonstration, using two different compilers.

Though the MRTSI was not originally intended as a standard, there has been some interest in using it as a starting point for a standard. Ideally, such an interface would guarantee interoperability of compilers and runtime environments, permitting the reuse of Ada runtime system implementations across compilers. The present MRTSI draft is too loosely specified to achieve this. It could be tightened up enough to achieve interoperability, but then it would probably need a specific variant for each target processor and memory architecture. This does seem practical from a technical point of view.

4.2 Impact on reuse

The main benefits of the MRTSI, or any common compiler-RTE interface, on software reuse for realtime applications would be through:

1. reuse of Ada RTE components across compilers;
2. reuse of applications which require special RTE features, through replacement of the compiler-provided RTE by a user-provided RTE (which includes the required special features).

An important special case of item (1) is where Ada is used as the implementation language for an operating system. There is then danger of a circular interdependency of the language and operating system implementations. A well defined compiler-RTE interface is essential if the operating system implementation is to be portable across different Ada compilers.

4.3 Other benefits of commonality

Recent experiences of MRTSI task force members have brought out other supporting arguments for adoption of an MRTSI-like interface, which go beyond issues of software reuse. Even very large users (e.g. IBM) have experienced intolerably long time delays in obtaining needed compiler/runtime system tailoring from compiler vendors, due to red tape or vendor manpower shortage. A MRTSI-like interface would allow the user to make such changes without delay. Increasing reliance by compiler vendors on "generic" (i.e. machine-independent) Ada runtime system implementations, in order to support increasing numbers of target architectures, is resulting in worsening performance. This aggravates the need for some customers to develop and maintain their own runtime system implementations, designed to work efficiently on their specific architectures. Some development activities requiring special runtime support are using compilers from more than one vendor for a single application, simultaneously or sequentially over the life of the project. Differences in capabilities of the RTEs provided by compiler vendors are a problem. With a clear interface, equivalent RTE implementations could be provided for the various compilers.

4.4 Future developments

Several difficult or complex issues were intentionally omitted from consideration for the MRTSI document, in order to make progress. These include: lower-level RTS details (e.g. stack frames, exceptions); an example of a machine-specific binding (e.g. 1750A); debugging support; I/O support; support for CIFO extension; support for priority ceiling protocol; multiprogramming support; multiprocessor support; support for distributed systems; support for systems with multiple address spaces (e.g. virtual memory).

The MRTSI task force expects to consider RTS interface proposals covering such extensions in the future, but as separate documents rather than revisions to the MRTSI. This is consistent with the goal of maintaining each document as a stable target for trial implementations.

The MRTSI task force also intends to begin a study of the POSIX realtime extension, and in particular a proposal for the inclusion of multithreaded processes, with the intention of providing feedback to the POSIX realtime extension working group. Where Ada is implemented on top of a POSIX-compliant operating system, the POSIX realtime extension is likely to be an important Ada RTE-internal or even compiler-RTE interface.

5 POSIX Ada binding

5.1 Background

POSIX is a standard interface between an operating system and applications. It is based on the UNIX system-call interface, but "there are no known Historical Implementations ... that will not have to change in some area to conform to the standard, and in a few areas the standard does not exactly match any existing system interface."[5]

POSIX is based on UNIX, and UNIX is thoroughly intertwined with the C language. UNIX implementations are generally written in C and built on top of the C runtime system. Applications interface to UNIX through C function calls. For these reasons, all POSIX interfaces are being drafted first as C-language bindings.

POSIX presently comprises a C-language interface for a set of basic system services, which is an official standard (IEEE Std. 1003.1-1988), and several extensions and language bindings, which are in various stages of development. The groups working on these extensions and language bindings are designated by numbers (1003.2, 1003.3, 1003.4, 1003.5, etc.). Working groups are added and reorganized from time to time. The two groups whose work is described here are 1003.4 (realtime extension) and 1003.5 (Ada language binding).

Via the extensions, POSIX is sprouting off a multitude of subsidiary standards, covering a range of subjects including security, networking, databases, system administration, and realtime computing. There is some danger that this proliferation of standards may get out of hand, but it may also lead to a more compatible and closely integrated set of standards for the full range of computing activities than has been seen before.

The POSIX standard is likely to have as great an impact on software development (and reuse) as the Ada programming language standard. In fact, it may have a larger impact, since it addresses needs of a larger community. If so, Ada programmers will want to use these interfaces.

5.2 The importance of an Ada binding

Where POSIX and Ada are used together, Ada applications will need an Ada interface to POSIX. More specifically, several organizations within the U.S. Department of Defense apparently need a standard operating systems interface, and have adopted or plan to adopt POSIX. However, these same organizations are required to use the Ada programming language. This gives some urgency to producing an Ada interface to POSIX, including the extensions.

Presently, Ada programs on UNIX systems use the pragma INTERFACE to call C functions. This imposes some extra computational overhead, but that is unavoidable when crossing between such dissimilar languages. The more serious problem, that could be corrected by providing a standard POSIX Ada binding, is that the correctness of the Ada application code at the interface depends on the data representation conventions of the Ada compiler, and on details of the Ada RTE implementation. Thus, if there are only C-language bindings for POSIX standards, C may turn out to be a more practical language for writing portable/reusable software

than Ada. This would be sad, since C is a less structured and more dangerous[2] language, which does not enforce interfaces as reliably as Ada.

Work on other language bindings (at least FORTRAN and Ada) is under way and, due to strong international pressure, there are also plans to produce "language-independent" versions of these standards at some point. This work is lagging a year or more behind the development of C-language standards, and is working at a disadvantage. Not only are the language binding efforts in a perpetual catch-up position, they are also hampered by some rather deep C language influences in POSIX. An example of a such a C influence is POSIX's assumption that something like C's *setjmp* and *longjmp* instructions are available. Without these it not possible for POSIX signal handlers to affect the main program's flow of control.

The POSIX 1003.5 working group has been working on developing an Ada interface to basic POSIX (1003.1), called the POSIX Ada binding. The group consists of about ten active volunteers, and has been at work for approximately two years. Progress has been very slow, due to the limited amount of time the group members have been able to spare from their paying jobs. As of September 1989, the draft is nearly complete. It is still missing one chapter, and some rationale, and has yet to be reviewed outside the working group.

It is important that there be Ada versions of all the POSIX interfaces. Despite the problem C-language influences, Ada bindings for most of the POSIX features can probably be produced. However, given the present rate of progress and the level of support this activity is getting, Ada bindings for the extensions are likely to be a long time in coming. The Ada binding group intends to get to the extensions eventually, but has not yet finished with basic POSIX. Still, progress on the POSIX Ada binding so far is encouraging. With some more skilled effort and some continuity of personnel this work is also likely to go faster as time goes on.

5.3 Overview of features

The draft POSIX Ada binding (Version 3.1.2) is a set of Ada package specifications with associated semantics. It attempts to provide Ada applications with the capabilities equivalent to those that are provided to C-language programs by IEEE Std. 1003.1-1988. These capabilities include:

1. Creating new processes (packages POSIX_Process_Primitives, and POSIX_Unsafe_Process_Primitives).
2. Sending signals between processes (package POSIX_Signals).
3. Identifying processes (package POSIX_Process_Identification).
4. Accounting for process time (package POSIX_Process_Times).
5. Checking and updating process environment variables (package POSIX_Process_Environment).
6. Changing the working directory (package POSIX_Working_Directory).

[2] For example, consider the statement "i = i/*p".1

7. Checking file permissions (package POSIX_Permissions).
8. Traversing a directory (package POSIX_Directories).
9. Creating and removing files (package POSIX_Files).
10. Checking the status of a file (package POSIX_File_Status).
11. Checking file limits (package POSIX_Configurable_File_Limits).
12. Opening, closing, and and manipulating files (package POSIX_IO).
13. Locking files (package POSIX_File_Locking).
14. Interacting with terminals (package POSIX_Terminal_Functions).
15. Obtaining information about users
    (packages POSIX_User_Database and POSIX_Group_Database).

5.4 Evaluation

These packages complement the standard runtime support available in Ada. By providing traditional operating system services, such as loading and executing programs, creating and updating permanent files, interacting with users over terminals, and accounting, a standard POSIX Ada binding will allow writing a wider variety of portable applications than can be written using Ada alone.

POSIX will support cooperation between multiple programs, including programs written in different languages. POSIX allows programs to initiate the execution of other programs and to communicate with them via files and signals. This will encourage the reuse of entire programs. Multiprogramming is especially important for long-running adaptable Ada systems, since each Ada program is a static entity. Multiprogramming therefore offers the only possibility for dynamic on-line reconfiguration of an Ada system.

Because POSIX is a standard interface supported by several widely available commercial operating systems, it is a realistic substitute for promised portable special-purpose Ada software development environments. There has been a lot of talk about Ada development environments and interface standards for Ada tools, such as CAIS[13] and the European Community's PCTE. So far, this talk has not made much of an impact on practice. The outside view of this activity are summed up in the words of one POSIX participant not involved in Ada, "For years, there's been a lot of talk about 'Ada environments', all of which seem, from a UNIX perspective, like enormous, cumbersome projects that might actually come into widespread use in, if not our children's lifetimes, perhaps their children's."

In contrast to future Ada environments, POSIX is practical and available today. POSIX provides the basic functionality needed to write portable software tools (along with other applications) in a simpler and less cumbersome form than other proposed interface standards for development environments. If more specialized support for Ada is required, POSIX may be used as the basis for a portable implementation of some existing interface, such as CAIS. It

may also make sense to evolve standard extensions to POSIX for language-specific interfaces, such as data interchange formats and symbolic debugging information.

For Ada implementors, the basic POSIX features (1003.1) are not enough to write a portable Ada RTE. Here we are talking about the C-language binding, since existing POSIX-compliant operating systems are based on C. Under these circumstances an Ada binding would not help, since it, too, would need to be built on top of a C-language interface. Until operating systems are written in Ada, and support for Ada tasking and I/O are built into the core of the operating system, non-embedded Ada RTE implementations will depend on services obtained via a foreign-language interface. It is therefore desirable for Ada language implementors that the C-language interface to POSIX provide some features that are currently missing, such as support for memory allocation. The POSIX realtime extension is especially important in the respect that it may help provide some of these missing features.

6 POSIX realtime extension

6.1 Background

The proposed draft POSIX realtime extension (1003.4) adds facilities for realtime applications to the features in IEEE Std. 1003.1-1988. As of September 1989, the POSIX realtime extension document is in its eighth draft, having gone through one "mock" ballot. It may reach formal ballot by early 1990.

The POSIX realtime extension goes beyond 1003.1 in more ways than functionality. More or less, 1003.1 standardizes conventional practices of the family of operating systems derived from AT&T's UNIX. Thus, the implementability and usability of the interface are well established, and conformant operating systems were already available by the time that standard was approved. In contrast, there are no well-established conventions for realtime support. Individually, the details of the 1003.4 interfaces are new, though they are based on features which have been implemented in various realtime variants of UNIX. Moreover, when all the features are taken together, they form a combination for which we have no experience with implementation or use.

The Ada programming language had similar origins; it, too, was a new combination of supposedly "proven" features, tested individually in other languages. As it turned out, there were enough differences and unforeseen interactions that it took several years of refinement before Ada was ready for practical use. However, it is possible that the POSIX realtime extension will not suffer from such a prolonged birth, since the POSIX realtime extension design process appears to have had much stronger participation by implementors than the Ada language-design process.

6.2 Overview of features

Draft 8 of the POSIX realtime extension includes the following features:

1. Binary semaphores. These provide low-overhead mutual exclusion between processes.
2. Process memory locking. This permits a process to request that certain regions of its code and data always remain in physical memory, thereby avoiding potential delays due to transferring information between memory and backing store.
3. Shared memory. This allows a process to share specified regions of memory with other processes.
4. Priority scheduling. This provides two "unfair" scheduling policies for realtime processes, which insures that they will not be slowed down by less critical processes.
5. Asynchronous event notification. This is a safe, queued mechanism for signaling events to processes; it addresses several deficiencies of the traditional UNIX signal mechanism, including the potential for lost signals.

6. Interprocess communication. This is a queued message-passing mechanism, which is integrated with the asynchronous event notification.

7. Timers. These offer multiple timers, different timer types, and much greater precision than the time reference available under IEEE STd. 1003.1-1988.

8. Synchronized input and output. This permits an application to insure that data is written to permanent storage immediately, rather than buffered in memory.

9. Asynchronous input and output. This permits an application process to continue execution while I/O operations are taking place. It is integrated with the asynchronous event notification scheme.

10. Realtime files. This is a complex scheme by which an application might negotiate with the operating system to obtain better performance out of file operations, by choosing from a set of options (which may or may not be supported by each system). Examples of such options include contiguous allocation, pre-allocation, and buffering.

11. Threads. These are multiple "light weight" threads of control within a process, similar in concept to Ada tasks. They all share one logical address space and one set of open files.

6.3 Support for Ada tasking implementation.

Unlike the basic POSIX features (1003.1), it appears that the realtime extension may be a sufficient basis for an implementation of Ada tasking. There are two ways this might be done: (1) each task might be a separate POSIX process; (2) each task might be a separate POSIX thread within a process. If a process-per-task model is used, the relevant POSIX features are: (a) shared memory, and the operations to manage it; (b) efficient interprocess synchronization (e.g. semaphores); (c) priority-based scheduling. On the other hand, if a process-per-program model is used, the main relevant feature is multithreaded processes.

If it is possible to produce a portable Ada language implementation based on POSIX with the realtime extension, this would certainly make it easier to port Ada applications. This would also provide programmers with more predictability and control over their applications. The POSIX realtime scheduling policies are more precisely defined than Ada's, and a programmer with direct access to the POSIX interface could take more direct control over scheduling and resource allocation decisions than is now possible with standard Ada alone.

There are several questions that remain to be resolved. First, it is not yet clear whether the 1003.4 working group will be able to reach a consensus on multithreaded processes. There are many technical difficulties, and some divergence of opinion, so that threads may not be part of the final standard. If threads are included, it is still not clear whether they will be defined in a form that will support the requirements of Ada tasking.

The draft POSIX Ada binding associates processes with Ada programs, rather than tasks. This decision seems to be the only way to avoid a conflict between two essentially incompatible models of concurrent programming. Perhaps the biggest difference between the two models is that POSIX processes do not share the same address space, or the same set of open files. The Ada language definition, says nothing about how main programs are invoked, or how they

may interact with one another. POSIX is therefore free to define the semantics of processes, so long as each process is a separate main program execution. This nice division of domains between POSIX and Ada will no longer apply if POSIX is extended to support multiple threads of control within a process. Then, there will be an overlap POSIX threads and Ada tasks. If there is a good match between these two, threads could be a big help to Ada implementors, but any semantic differences between POSIX threads and Ada tasks could cause trouble.

If the POSIX realtime extension supports multiple threads, any future Ada interface will have to decide whether to make the POSIX thread operations visible, or to only let Ada applications use the standard Ada tasking operations. If the thread interface is exposed, it may offer a useful alternative to Ada tasking, but exposing it is likely to be unsafe if the Ada language implementation uses it to implement Ada tasks. In the latter situation, an Ada application calling the POSIX thread services directly could interfere with the Ada tasking implementation. This is likely to mean that the more useful the POSIX realtime extension is for implementing Ada on top of a C-language operating system, the less an Ada application can be allowed to access them directly. In short, two outcomes are likely: either Ada tasks will be POSIX threads, with only the standard Ada tasking operations, or POSIX threads will exist as an alternative to Ada tasking, that cannot be mixed.

If multithreaded processes are not included in the POSIX realtime extension, the only way to use the realtime extension to implement Ada tasks will be via a process-per-task model. However, this conflicts with the underlying model of the current draft POSIX Ada binding, which is a virtual-process per program. This means that if the Ada language implementation does implement a single Ada program as multiple POSIX processes, the POSIX Ada interface must hide the existence of all but one of these processes from the Ada program.

No matter whether POSIX ends up supporting threads, and whether Ada tasks will be implemented as threads, there remain technical questions concerning the adequacy of the POSIX realtime extension to implement Ada tasks. Moreover, there is the question of whether using the POSIX realtime extension to implement Ada tasks is even desirable, since it might restrict access by Ada programs to the full functionality of POSIX.

6.4 Substitute/extension for Ada tasking

The POSIX realtime extension may provide useful features that are not found in Ada, or which are more powerful or have more precisely defined semantics than the corresponding Ada features. For example, POSIX realtime timers provide more precision and greater functionality than the Ada `delay` statement and `CALENDAR.clock`. POSIX realtime process scheduling is more precisely controllable than Ada task scheduling. Also, the POSIX event and message-queue mechanisms offer functionality similar to the Ada rendezvous, that is more flexible and potentially more efficient. More specifically, they provide a form of asynchronous communication that seems to be needed in realtime systems, and which Ada has been criticized for lacking.

POSIX also may provide more appropriate abstractions than Ada. The semaphores, events, and messages provided by the POSIX realtime extension is closer to conventional models of concurrent programming than Ada's rendezvous. Moreover, it can be argued that by providing

several distinct standard communication object types (as compared Ada, where the programmer must construct equivalent *ad hoc* substitutes using intermediary tasks), the POSIX realtime extension encourage the programmer to design at a higher level of abstraction.

Application programmers will not want to do without useful POSIX features, just because they are programming in Ada. It is therefore important to develop an Ada binding for the POSIX realtime extension that passes through any advantages POSIX may offer – i.e., that extends the functionality of Ada in a compatible way.

It is still too early to see how well the POSIX realtime extension can be interfaced to Ada, since the C-language binding is still evolving, and work on an Ada binding has not yet begun. If the POSIX realtime extension cannot be well integrated with Ada tasking, Ada programmers in a POSIX environment may end up working directly with the POSIX process and thread abstractions, instead of Ada tasking. That is, the POSIX realtime extension may force a choice between POSIX and Ada, or at least a choice between a POSIX model of concurrency and Ada's.

6.5 Prototype implementation

As part of this project, we had sufficient resources to do some experimentation with Ada runtime interfaces. We chose to apply our efforts to implementing, in prototype, as much of the draft POSIX realtime extension as we could, since the POSIX realtime extension is a draft in C-language form, with no work yet started on an Ada binding and no complete implementation. A prototype implementation could be helpful in providing timely feedback on the draft C-language binding, and could speed up the future development of an Ada binding. A prototype implementation would also be useful as a basis for evaluating both the direct utility of the POSIX realtime extension and its potential for implementing Ada tasking.

The only practical way to produce a rapid prototype of the POSIX realtime extension was to start with an existing UNIX implementation. We chose to start from the Sun Microsystems UNIX (SunOS) operating system, because we had the source code and the computers to compile and run it. Since SunOS is written in C, this meant programming in C. The work involved modification of the SunOS kernel and other portions of the operating system.

The POSIX realtime extension prototype includes (in order of completion):

1. scheduling and priorities (Section 6 of 1003.4);
2. semaphore special files (Section 3 of 1003.4);
3. shared memory special files (Section 5 of 1003.4);
4. a restricted form of timers (Section 8 of 1003.4).

The motivation behind these choices is that these features seemed to be a minimum set needed to begin experimentation with realtime programs. They also appear to be a minimum set necessary to provide functionality similar to Ada tasking, and perhaps to serve as a basis for a one-process-per-task implementation of Ada tasking.

The prototype implementation attempts to follow the draft POSIX realtime extension, but is not completely compliant. The differences are generally minor. They are mostly due to our desire to fit the extension within the framework of the basic SunOS implementation, and in particular our need to stay within the limitations of the Sun Network File System (NFS) since the development and testing could only be done on a diskless machine.

A more detailed discussion of the prototyping of these three features is presented in Appendix A.

6.6 Evaluation

It is still too early to make firm conclusions about the POSIX realtime extension. Not only is the standard still only in draft form, but there is little or no experience on which to base an evaluation. Our prototype implementation only added a little to this.

It is not yet clear whether POSIX can be implemented efficiently enough to meet the needs of hard realtime applications. However, it is likely to be very useful for nonembedded systems where realtime and nonrealtime processes run on the same computer. It is also likely to be very useful for prototyping realtime systems, on a development host.

Based on our prototype implementation, the POSIX realtime extension appears to offer a dramatic improvement in speed, predictability, and utility over both basic POSIX (1003.1) and the UNIX system on which our prototype was based. In short, the extension permits writing portable realtime applications — a thing which cannot be done in basic POSIX or UNIX. Using the features in our prototype we were able to obtain reliable control over execution timing down to the 20 millisecond range[3].

One of the main weak-points of the draft POSIX realtime extension is its heavy reliance on the POSIX file-system name space for semaphores, shared memory, message queues, and all other named objects. Since there is no type distinction between different kinds of file descriptors, all operations must be implemented so as to be safe for all file types (e.g. a semaphore wait operation might be applied to a pipe or terminal device, erroneously). For the same reason, essentially all operations will require switching to kernel mode. (Our experiments with a prototype implementation lead us to believe this amounts to about 100μs overhead per system call.) For these and other reasons the utility of the POSIX realtime extension is likely to be limited to applications with fairly loose timing constraints.

POSIX is a general purpose operating system, so support for realtime applications requires compromises. One of these compromises is some extra overhead for realtime applications, to support features that are not ordinarily required in a pure realtime operating system. This overhead will be noticeable, but it is not yet clear how far it can be reduced by heuristics, or what range of applications will be able to tolerate it. Non-realtime applications are also compromised. In order to provide timely service to realtime processes, other processes may be denied service.

The POSIX realtime extension takes an approach to scheduling that beats Ada three ways. POSIX provides a choice of two well-defined scheduling policies (FIFO and round-robin), and

[3]The resolution of the Sun workstation clock.

permits the policy and priority to be changed dynamically on a per-process basis. In contrast, (1) Ada's scheduling policy (priorities) is not completely defined, (2) it provides no choice of scheduling policy, and (3) it does not permit scheduling policy or parameters (priority) to be changed. Both POSIX and Ada allow implementations to provide nonstandard scheduling policies, but POSIX provides options in the standard interface for selecting nonstandard policies, while Ada specifies nothing.

The POSIX realtime extension also standardizes a higher level of support for realtime applications than Ada does in other areas. Among these areas are timer precision, control over synchronism/asynchronism of I/O, and asynchronous communication. Like Ada, the POSIX realtime extension does not impose absolute performance requirements, but it does improve on Ada by defining performance metrics for its features.

All this does not mean POSIX is necessarily "better" than Ada tasking for hard realtime applications. First, Ada is a language, and POSIX is an operating system interface. It should not be surprising if the POSIX realtime extension is a better operating system interface than Ada. (Certainly, it is not surprising that Ada is a better language than POSIX!) Second, we are comparing standards rather than implementations. As a standard, POSIX requires more RTE services than Ada. Of course, Ada implementations designed for embedded realtime systems offer implementation-defined features which go beyond the Ada standard. For embedded applications, Ada implementations are likely to be able to offer far better performance than any POSIX-compliant realtime operating system, because they do not have to support all of the general-purpose operating system features that POSIX does.

Those responsible for the Ada 9X revision can learn from the POSIX realtime extension. At least, they can learn to provide both more concrete default semantics and implemention-defined options, for details Ada currently leaves to the language implemenation. They may also find some specific capabilities that might be added to Ada, but perhaps this should be left to secondary standards like POSIX.

It remains to be seen whether an Ada interface to the full POSIX realtime extension can be provided that is compatible with the Ada tasking model. This issue deserves more attention as the POSIX realtime extension solidifies. If our prototype implementation is extended further, it may be helpful as a tool for answering this question.

7 Conclusion

We have examined the role of the Ada language and common package interfaces for the Ada runtime environment in the development of reusable software, especially for realtime applications. More specifically, we have looked in depth at the MRTSI developed by ARTEWG, the POSIX Ada binding, and the POSIX realtime extension. For evaluation purposes, we have developed a partial prototype implementation of the POSIX realtime extension, including the realtime scheduling and semaphore-management functions.

Though the Ada language supports software reuse, there are several ways in which Ada might be revised to improve it in this regard. We have pointed some of these out.

One of the ways Ada supports reuse best is via package interfaces. Agreement on common package interfaces for frequently needed components would promote reuse of both those components and applications which rely on them. Such package interfaces can serve as bridges between reusable and nonreusable code, enabling components that depend on them to be reused even when the package bodies cannot.

A standard compiler-RTE interface, like the MRTSI, could promote reuse of Ada software, if commonly adhered to. Compilers and RTE implementations would be more reusable. In particular, realtime RTE implementations could be reused across compilers. Since realtime applications typically must depend on implementation-specific features of the Ada RTE, this would in turn promote reuse of realtime applications.

The POSIX standard interfaces also could promote reuse of Ada software. They provide a standard interface for operating system services that are required by many Ada applications, but which are not provided by the Ada language. Without a standard Ada interface, Ada programs requiring these services cannot be reused without code changes; with a standard interface, they can be. Work on Ada versions of these interfaces is therefore important.

The basic POSIX interface (1003.1), for which the Ada binding is currently being developed, provides services that are mostly needed by nonrealtime applications, including Ada compilers and software development tools. In contrast, the POSIX realtime extensions can directly benefit realtime applications. In addition to directly supporting realtime applications, the realtime extension may support portable Ada RTE implementations.

How much the POSIX realtime extension helps realtime applications will depend on how it evolves between now and the time it is formally adopted as a standard, and how efficiently it can be implemented. Based on our partial prototype, it already provides enough functionality to write portable realtime applications with timing constraints measured in tens of milliseconds. While this is far below the capabilities of the best embedded realtime Ada implementations, it is far better than the worst permitted by the Ada standard.

The POSIX realtime extension provides more standard support for realtime applications than the Ada language does. However, POSIX also provides many general-purpose operating system features that are not required by realtime applications. Supporting all this will undoubtedly force POSIX implementations to compromise on the efficiency of their support for realtime. It remains to be seen how wide a range of realtime applications will be able to tolerate this

compromise.

The strengths of the POSIX realtime extension will probably turn out to be: application portability; the ability to mix (soft) realtime applications with ordinary applications; the ability to prototype embedded realtime applications in a convenient development environment, without need for tedious cross compilation, downloading, and cross-debuggers.

Organizations committed to using Ada should beware of the strong linkage between POSIX and the C programming language. Using POSIX as an operating system interface will bring pressure to use C as a programming language. This pressure will be aggravated by the time lag between the introduction of C-language bindings for POSIX and its extensions and the development of Ada bindings. A short term solution to the lack of Ada bindings to write Ada applications using pragma `INTERFACE` to access the C-language POSIX interfaces. However, Ada code near the interface is likely not to be portable across compilers. Of course, there is the option of switching from Ada to C, but that decision should be made on the merits of the language (where Ada would probably win) rather than indirectly through the choice of operating system interface.

Issues needing further study. Based on our study, the following issues appear to need further study:

1. *Testing of the MRTSI.* The concept of a standard compiler-RTE interface needs testing. (As mentioned in Section 4.1, this issue is apparently being addressed by the U.S. Air Force.)

2. *RTE-internal interfaces.* Standard "bridging" interfaces for internal RTE modules that are most likely to need application-specific tailoring could greatly improve the reusability of the rest of an Ada RTE implementation. Suitable internal interfaces should be identified, and the concept tested. Possible candidates include the timekeeping and task scheduling functions.

3. *Ada binding for realtime extension.* An Ada binding needs to be developed for the POSIX realtime extension. Work should start right away, rather than waiting until the Ada binding for basic POSIX (1003.1) is complete.

4. *Evaluation of realtime extension.* A more complete prototype of the POSIX realtime extension should be developed and tested. (Work along this line appears to be going on among UNIX vendors.)

5. *Ada on realtime extension.* An attempt should be made to implement an Ada RTE using the features of the POSIX realtime extension. Ideally, both the process-per-task and thread-per-task model should be explored.

References

[1] U.S. Department of Defense, *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A, Ada Joint Program Office (January 1983).

[2] U.S. Department of Defense, "Department of Defense Requirements for High Order Computer Programming Languages", *SIGPLAN Notices*, 12,12 (December 1977).

[3] U.S. Department of Defense HOLWG, "Department of Defense Requirements for High Order Computer Programming Languages: 'Steelman"', (June 1978).

[4] J. D. Ichbiah, J.G.P. Barnes, R.J. Firth, M. Woodger, *Rationale for the Design of the Ada Programming Language*, Alsys S.A. (1986).

[5] *IEEE Standard Portable Operating System Interface for Computer Environments: IEEE Std 1003.1-1988*, IEEE (1988).

[6] SIGAda ARTEWG, "A Model Runtime Environment Interface for Ada", *Ada Letters* IX, 1 (January/February 1989) 84-132; also published in *Ada User* 10, Ada Language UK, Ltd. (special issue) (1989).

[7] ARTEWG, Ada Run Time Environment Working Group(ACM SIGAda), "A Catalog of Interface Features and Options for the Ada Run Time Environment", Release 2.0, ACM SIGAda (1988).

[8] CECOM Center for Software Engineering, "A Catalog of Ada Runtime Implementation Dependencies", CECOM Center for Software Engineering, Report CIN: C02 092JB 0001 (February, 1989).

[9] CECOM Center for Software Engineering, "Transportability Guideline for Ada Real-Time Software", CECOM Center for Software Engineering, Report CIN: C02 092LA 0008 (May, 1989).

[10] T.P. Baker, "A Corset for Ada", technical report TR-86-09-06, Computer Science Department, University of Washington, Seattle, WA (1986).

[11] T.P. Baker and K. Jeffay, "A Lace for Ada's Corset", technical report TR-86-09-05, Computer Science Department, University of Washington, Seattle, WA (1986).

[12] T.P. Baker, "A Low-level Tasking Package for Ada", *Using Ada: ACM SIGAda International Conference*, Boston, MA (December 1987) 141-146.

[13] U.S. Department of Defense, *Military Standard Common APSE Interface Set (CAIS)*, Proposed MIL-STD-CAIS, Ada Joint Program Office (January 1985).

[14] Third International Workshop on Real-Time Ada Issues, Nemacolin Woodlands, PA (May 1989); proceedings to be published in Ada Letters.

[15] Ada 9X Project Requirements Workshop (June 1989), Ada 9X Project Report, Office of the Under Secretary of Defense for Acquisition.

A Report on prototyping activity

A Prototype POSIX Realtime Extension

Jim Hudgens and Jim Groh
the Florida State University

This appendix describes some of the issues encountered during the course of work on prototyping portions of the proposed POSIX realtime extension (1003.4) for the IEEE standard Portable Operating System Interface for Computer Systems (POSIX). The four features implemented are: (1) scheduling; (2) binary semaphores; (3) shared memory; (4) realtime timers. The latter two were only partially implemented.

This work was carried out by modifying the C-language source code of the SunOS (V4.0).

A.1 Scheduling

The implementation of realtime process scheduling was rather interesting. It was relatively difficult to design a match for the POSIX semantics within the context of the SunOS code. However, once the design work was done, the actual changes were relatively small. Aside from the actual implementation of new system calls, for scheduling, very little code had to be modified to achieve a compliance with the POSIX realtime extension.

The POSIX realtime extension provides a specific set of interfaces which allow a user program to directly (and almost deterministically) control its allocation of CPU time. In the normal SunOS a program cannot do much to affect its allocation of CPU time; it can only change its "nice" value.

The changes required to implement the extension involved:

1. Modifying the priority mechanism so that the priority of a process can be controlled by the application program instead of the kernel scheduler.
2. Changing the mechanism by which the runnable processes are queued and dispatched.
3. Changing the manner in which a preemption is handled.

The normal scheduling paradigm under UNIX is generally described as a prioritized multi-queue round-robin scheme. It has the following characteristics:

1. Periodically (once every second, for SunOS) a running process has its priority modified, based on its current usage of the CPU. This is done in the routine *schedcpu*. The process's position in the run queues is modified, if it is runnable.

2. Periodically, (once every 100ms, for SunOS) a kernel thread awakens and sets a flag which is used to force a context switch. This is done by the routine *roundrobin*. The process which is running at that point is preempted upon return to user context. The effect is to force the processes at each priority to alternate execution in a round-robin manner.

3. If a process blocks on a system call, its priority is raised (to a more urgent priority). This does not change under the POSIX realtime extension.

4. When the kernel dispatches a process to the CPU, it always chcoses the runnable process with the highest priority. This dispatching occurs in the routine *swtch*.

5. When a running process makes a switch from user to kernel mode, if it does not block in the system call (generally via *sleep*), it may be preempted (inside routines *trap* and *syscall*) when attempting to return to user mode if there is a higher priority process which is runnable.

The POSIX realtime extension does not follow this paradigm. In particular, realtime POSIX processes are not all treated identically by the scheduling system. A set of new system calls have been added which allow a process to identify itself to the operating system as a realtime process.

The new process types are defined in the following way. (This is paraphrased from a draft of the POSIX realtime extension.)

FIFO processes:

- When a FIFO process becomes blocked and is set runnable, it becomes the newest process on the list for that priority.
- When a FIFO process is preempted, it becomes the oldest process on the list for that priority.
- When the priority of a FIFO process is changed, it becomes the newest process on the list for that priority.

Round-robin processes:

- This is identical to FIFO except that at *RR_INTERVAL* intervals, a round-robin process is replaced on the list for that priority at the newest position.

In implementing the above specifications we had to make the following changes to the SunOS implementation:

1. The routine *schedcpu* still recalculates the priority of all processes; however, it does not change the priorities of processes which are marked as realtime. This means that a process

which is to be treated as a realtime process must be marked as such in its process structure. We used two flag bits to represent the realtime status of a process:

00 = *SCHED_OTHER*
10 = *SCHED_RR*
11 = *SCHED_FIFO*

If the high bit (common to both *SCHED_RR* and *SCHED_FIFO* processes) is set, then the process is treated as a realtime process, and its priority is changed only via system call; the dynamic kernel priority mechanism is disabled for this type of process.

2. The routine *roundrobin* still runs periodically at 100ms. It still sets the flag for a context switch, regardless of the type of the process currently executing.

3. When a process makes a system call, if it is a realtime process, its priority is changed, but only if that change increases the priority.

4. The *dispatch* function of the kernel is identical; no changes were necessary.

5. When a process attempts to return to user context, it may be possible that the system considers it to be preempted. The treatment here is special, because if it is a FIFO or a round robin process being preempted, then it should go back to the front of the run queue. This is handled by the code in `trap.c` and in the assembly routine `vax.s`. In `trap.c`, in the routines *trap()* and *syscall()*, code was added to handle the case where the current process is to be preempted, and it is either a FIFO process, or it is a round robin process and its time slice has not expired (this is approximated). In either case, an additional bit is set in the process flags, which tells the assembly routine *setrq()* to enqueue this process at the front of the run queue for that priority.

6. Four system calls were written which change the process priority and the process scheduling algorithm. The range of SunOS internal priorities ranges from 0 (highest) to 127 (lowest). There are (under the current implementation) 32 run queues, with four priority levels assigned to each run queue. Since the semantics associated with the POSIX real time extension require that processes of different priority be in different queues, we can support at most 32 POSIX realtime priorities. Of the available 128 internal priorities, those in the range 0 to 24 (i.e. those reported to be in the range -25 to -1 by **ps**) are probably not safe to use for user processes, so we are left with a usable internal priority range of 127 to 25, and a usable POSIX realtime priority range of 0 (*PRIO_MIN*) to 25 (*PRIO_MAX*). [4] Thus, the translation between POSIX and internal priorities may be expressed in C macros, something like:
#define INTERNAL_TO_POSIXRT(x) (31 - (x)/4)
#define POSIXRT_TO_INTERNAL(x) (127 - (x) * 4)

7. The system calls *rt_setpriority* and *rt_setscheduler* were carefully written to conform to the 1003.4 semantics, which deal with the sequence of events which happen when a process sets the priority of another process.

8. The *fork* system call had to be changed to force the above semantics to be inherited across a *fork* call.

[4] The order of the two ranges is reversed intentionally, to reflect the difference between the interpretations of POSIX and SunOS internal priorities. Also, the ranges of normal process priorities and realtime process priorities overlap, as is permitted by POSIX.

The result of this effort is a scheduler which has very different characteristics from the normal scheduler. However, without a synchonization mechanism it is difficult to predict or control process execution sequences, due to race conditions. In particular, the execution sequence associated with the creation of new processes seems to be particularly difficult to synchronize. Fortunately, this can be solved using semaphores.

A.2 Semaphores

A standard solution to the synchronization problems in a multiple process system typically involves semaphores and shared memory. It was for this reason that we attempted an implementation of the semaphore primitives as described in the POSIX realtime extension.

One major difficulty in attempting to implement the POSIX realtime semaphores is a result of the semaphores being identified in the filesystem name space. In particular, adding support for semaphores to the existing kernel code also required modification to existing code dealing with filesystems, and doing so in a manner which does not break existing filesystem functionality.

There were several major logistic problems in attempting to implement this set of system calls; the two biggest being lack of time and lack of a dedicated machine with disk drive to test the resulting kernel. For this reason, the implementation of semaphores is only partly correct. However, it is correct in the "spirit" of the POSIX realtime extension; the defects are mostly minor deviations from the 1003.4 specifications, and some rather inelegant kernel modifications. The current implementation of semaphores interfaces to the filesystem code is based on the SunOS version of the System V named-pipes. The difficulty is to get the files created and identified as "semaphore special files", without making any modifications to the network server filesystem code. The only low-level code which was changed dealt with NFS mounted files. None of the kernel code has ever been tested on machines having local disk drives, since none were available for this purpose.

The upper-level code for creating and accessing semaphores seems correct and fits cleanly into the rest of the filesystem code. All the system calls dealing with semaphores share the same entry point, through a routine called *rtsem()*. The system call for *rt_mksem* calls the file create operation, *vn_create()*, and passes a new attribute type: *VSEM*. The *vn_create()* routine in turn calls the filesystem specific routine, *nfs_create()*. The routine *nfs_create()* fakes the remote operation (actually creates a pipe), and then calls a special routine needed for special files, called *specfs()*. The routine *specfs()* deals with special files and devices, and when passed arguments for a *VSEM* file, calls the routine *semsp()* which allocates the *vnode* for the semaphore, and associated storage for the queuing of processes. The parts of this scheme which are inelegant involve the faked create in the routine *nfs_create()*. The outcome of this is that one cannot possibly get the semantics of a persistent semaphore (*SEM_PERSIST*), since there is nothing in the actual created file which identifies it as a semaphore, much less the actual status of the semaphore. It also appears that the semantics defining the last close of a semaphore (or files in general) might be incorrect, in that if a semaphore file is created with initial state *SEM_LOCKED*, it retains that state even though it is not held open by any process between the time of creation (via *rt_mksem()*) and the time of its first open.

Any of the calls which are passed the file descriptor of a semaphore special file then use the

default "descriptor to *vnode*" lookup routines.

In a call to *rt_semwait()*, if the semaphore is in state *SEM_LOCKED* the process enters a new process state, *SSEM*, and is queued on the semaphore *vnode* in ascending priority. A context switch then occurs. Otherwise the process locks the semaphore and continues execution.

In a call to *rt_sempost()*, if the semaphore is in state *SEM_LOCKED* but has no queued processes, the call allows the process to proceed and the semaphore is set to state *SEM_UNLOCKED*. If the semaphore is locked and has queued processes, the first process is dequeued and set to a runnable state, and the semaphore remains in state *SEM_LOCKED*. As a final note, these calls will only work for semaphore files which reside on NFS mounted filesystems. No disk based filesystem code was modified or included with this system.

The *rt_semifpost()* and *rt_semifwait()* calls behave similarly, except as described in the draft POSIX realtime extension document.

This implementation differs from the System V implementation in that:

1. It uses the file system.
2. The semaphores are binary.
3. The sleep mechanism involves only processes waiting on the same semaphore.
4. The wakeup mechanism is potentially much faster, because the processes are queued in ascending priorities.

However, it is possible that using the filesystem name space could be a real handicap in terms of speed of access. In this implementation no attempt was made to optimize the translation from file descriptor to the corresponding *vnode*, nor was any attempt made to determine if the implementation had comparable or better speeds than the corresponding System V calls.

A.3 Shared Memory

Another major component in a realtime application is a mechanism concurrent processes to share memory resources. This feature is included in the POSIX realtime extension. It is both conceptually simpler and easier to implement than message passing, which is also included.

Most of the same difficulties encountered in the implementation of semaphores were also encountered in the implementation of the shared memory. Additionally, we were hampered by a lack of general documentation on the kernel memory management routines, which tend to be somewhat hardware (vendor) specific. For these reasons, the implementation of memory system calls are only approximately correct, and not well tested.

The upper-level code for creating and accessing shared memory seems correct and fits cleanly into the rest of the filesystem code. All the system calls dealing with semaphores share the same entry point, through a routine called *rtshm()*.

The system call for *rt_mkshm* calls the file create operation, *vn_create()*, and passes a new attribute type: *VSHM*. The *vn_create()* routine in turn calls the filesystem specific routine, *nfs_create()*. The routine *nfs_create()* fakes the remote operation (and actually creates a pipe), and then calls a special routine needed for special files, called *specfs()*. The routine *specfs()* deals with special files and devices, and when passed arguments for a *VSHM* file, calls the routine *shmsp()* which allocates the *vnode* for the shared memory file, as well as allocating structures (*struct anon*) which are managed by the kernel memory management routines. When the call to *vn_create* returns, the actual memory is allocated.

When the file is mapped, most of the arguments are ignored. The only argument which is not ignored is the file descriptor. We let the system choose a virtual address at which to map the memory, and then map the entire memory into the process's address space. The code which does this is modeled after the code for System V shared memory. It consists of a call to the routine *map_addr()*, followed by a call to *as_map()*. So by ignoring most of the arguments to *rt_map()*, much of the semantics to this call are lost. In addition, the semantics associated with *exit* and *fork* may or may not hold. And as in the semaphore case, the parts of this scheme which are inelegant involve the faked create in the routine *nfs_create()*. The outcome of this is that one cannot possibly get the semantics of a *SHM_PERSIST*, since there is nothing in the actual created file which identifies it as a shared memory, much less the actual memory allocation. As in the case of semaphores, this call only works on NFS mounted partitions, which reflects the development environment.

We are certain that many aspects of the implementation are correct, such as the filename to memory mapping, and that the *rt_map* call actually does map in memory which can be shared with other processes. This is the functionality we needed to prototype and test systems of processes, and to write the demonstration program *BANDS_DEMO*.

A.4 Realtime timers

The realtime timer feature of the POSIX realtime extension was partially implemented, because some form of timer was necessary to write test cases for the other features. This is incomplete. Also, due to limitations of the Sun hardware (i.e. 10ms clock ticks), it does not meet the requirements for precision of the POSIX realtime extension.

A.5 Summary

The prototype implementation of a subset of the POSIX realtime extension appears to function correctly, and has proven adequate for experimental evaluation of the utility of the features which are supported. It also seems to be a useful tool for prototyping realtime applications on a UNIX development host.

B Demonstration program

BANDS_DEMO is a program that was designed as a tool to study the execution sequences of multiple processes in a priority-based scheduling system, using various scheduling algorithms. The program is designed so that each process can be described in terms of its priority, period, execution time, and the scheduling algorithm used. The output of the program is a graphical display, which is somewhat similar to a Gantt chart. The display is designed to show (in an approximate way) the execution sequence of the system of processes.

To utilize this software, you need the following:

1. A **diskless** Sun 3/50 or Sun3/60 running SunOS 4.0 or later.
2. A server server running SunOS 4.0 or later, having a quarter inch tape drive.
3. Our modified SunOS kernel.
4. The *BANDS_DEMO* source and/or binaries.

The *BANDS_DEMO* program is described in the *Postscript* document `BANDS_DEMO.ps`. The kernel modifications are described in `posix_rtkernel.ps`.

Included on the distribution tape are the following files and directories:

```
-rw-rw-r--  1 hudgens      5310 Sep  3 14:51 README
drwxrwxr-x  4 hudgens      1536 Sep 10 17:31 c-bands
drwxrwxr-x  7 hudgens      2560 Sep 10 22:41 ada-bands
-rw-rw-r--  1 hudgens     25612 Sep  3 14:51 BANDS_DEMO.ps
-rw-rw-r--  1 hudgens     25612 Sep  3 14:51 posix_rtkernel.ps
-rw-r--r--  1 hudgens    619467 Jul 31 09:00 vmunix
```

The `c-bands` directory contains a C-language version of the *BANDS_DEMO* program, as well as all necessary files required to make it. The `adabands` directory contains the corresponding version of the program in Ada. The `vmunix` file contains a bootable kernel. The other files are documentation which can be printed on any *Postscript* printer.

To use the system, do the following:

1. Locate a **diskless** Sun3/50 or Sun3/60 workstation.
2. Copy the contents of the distribution tape to this client's `tmp` directory. For example:

```
server% cd /export/root/client/tmp
server% mkdir bootdir
server% tar xvf /dev/rst8
.....
```

This step can be done in several other ways, using *rsh*.

3. Halt the client system:

```
client#  shutdown now
.....
# halt
```

4. Boot the client using the new kernel:

```
> b tmp/bootdir/vmunix
.....
.....
```

5. Log in and start the demo:

```
client% cd /tmp/bootdir
client% cd c-bands
client% BANDS_DEMO
.....
client% cd ../ada-bands
client% BANDS_DEMO
.....
```

The demo can be stopped using the interrupt character (i.e. control-C).

Read the detailed documentation (mentioned above, provided in *Postscript* form) before you attempt to use this demo. It includes examples of how to interpret the graphical output.

NOTE

The documentation and files for the BANDS_DEMO program are not included with this report. If you require further information on the demonstration program, please contact Mary Bender, 201-544-2105 (AV 995-2105)